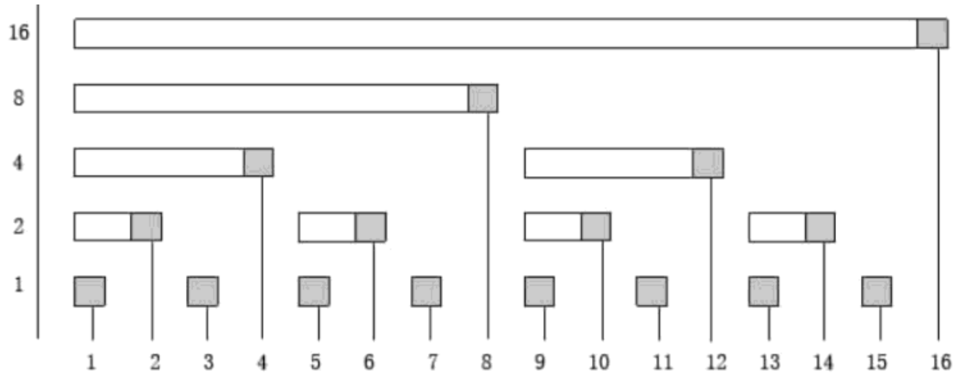
树状数组

2019年4月4日　　13:20

1. 树状数组binary indexed trees/Fenwick tree：维护序列前缀和的数据结构
    1) 适用于只需要求从头开始的前缀“和”的情况，或者有对应逆元的区间“和”（此处“和”指可累积的运算的运算结果）
2. 按lowbit(x)分解区间[1,x]为O（logx）个小区间
    1) 设x=0+2^a+2^b+......+2^m+2^n，则目标区间为
        i. 长度2^a的[0+1,2^a]
        ii. 长度2^b的[2^a+1,2^a+2^b]
        iii. ......
        iv. 长度2^n的[2^a+......+2^m+1,2^a+......+2^m+2^n]
        v. 即while(x>0){
            1) printf("[%d,%d]", x- (x&-x)+ 1, x);
            2) x-= x&-x;}
    2) 即上图中忽略大于x的绿点，从左上往右下每层的最长区间
3. 树状数组c的性质
    1) c[x]保存了以它为根的子树的所有叶节点的和
    2) c[x]的子节点数=lowbit(x)的位数
    3) c[x]父节点是c[x+lowbit(x)]
    4) x为2的整数幂时，树的层数为log(x)
    5) x不是2的整次幂时，层数是log(x)向上取整，不过此时其实是森林
4. 树状数组的前x项和查找
    1) int ask(int x){

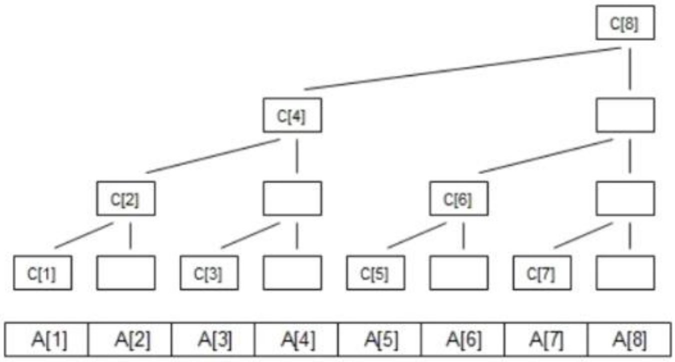
```
        int s=0;
        for(; x>0; x-=x&-x)
            s+=c[x];
        return s;}
```

    2) 

    3) 结合上图，是一层一层往“左上”累加的
    4) [l,r]之间的和=ask(r)-ask(l-1)
5. 单点增加a[x]时对c[x]的更新
    1) void add(int x,int d){

for(; x<=n; x+=x&-x)

c[x]+=d;}

2) 

3) 结合上图，是逐步往“右上”父节点更新

4) 可用于初始化树状数组，时间复杂度O（NlogN）

5) 如果用lowbit运算扫描所有子结点并求和，每条边只会被遍历一次，时间复杂度为O（k从1到logN Σ k*N/ 2^k）即O（N）

6. 结构体模板

```
struct BIT{
    int N,v[MN];
    void init(int nn){ N=nn; for__(i,0,N) v[i]=0; }
    ll sum(int x){ ll s=0; for(; x>0; x-=x&-x) s+=v[x]; return s; }
    void add(int x,int d){ for(; x<=N; x+=x&-x) v[x]+=d; }
}bit;
```

◆

◆ 例题

1. 计算逆序对

1) 用树状数组c[i]记录i在数组a中的出现次数，则ask(r)-ask(l-1)即为数组a中[l,r]范围的数量

2) 为了让从右往左求前缀和能算出结果，需要逆序扫描数组a，求比它小的数

3) rof_(i,n,0){ //逆序遍历，查a[i]后的每个数

cnt+=ask(a[i]-1); //比a[i]小的数字量

add(a[i],1); //更新a[i]的数量

}

4) 时间复杂度O（(N+M)logM），当最大数M较大时可能需要事先离散化，但离散化时需要排序，不如直接在归并排序时计算逆序对

2. 计算上升子序列数量

1) 先给序列排序，之后离线按升序逐个插入树状数组（数域打的话先离散化）

2) struct node{

int idx; //从0开始的a0的初始下标

ll v; //a0[idx]的值

bool operator<(const node r)const{

return v < r.v;

} //main函数中会按v升序给a0[i]排序

}a0[MN]; //排序后按i遍历会获得不降序列，靠idx获得初始下标

3) for_(idx,0,n)

ed[idx]= (bit1.sum(a[idx] -1) +1) %P,

//从最前到a[idx]-1为止的上升序列总数 + 1（a[idx]本身结尾） = 以idx结尾的上升子序列数

```
bit1.add(a[idx], ed[idx]);
```

4) 同理，反向遍历可得第i个数开始的上升子序列数量
5) rof__(idx,n-1,0)

```
bg[idx]= (bit2.sum(n) -bit2.sum(a[idx]) +1 +P) %P,
```

//从a[idx]+1到最后为止的上升序列总数 + 1（a[idx]本身开始） = 以idx起始的上升子序列数

```
bit2.add(a[idx], bg[idx]);
```

6) ed的总和=bg的总和=总上升子序列数量
7) ed[i]*bg[i]=包含第i个数的上升子序列数量